Paper JMP-xx

Run Program - JMP's Link To Other Programs

Michael Hecht, SAS Institute Inc., Cary, NC

Abstract

JSL's `Run Program()` function launches other programs, sends data and commands to them, and retrieves their output. With this powerful tool, script authors can extend the reach of JMP to drive all the capabilities of their machine. But harnessing this power can be challenging, even for the experienced script author. Several examples of `Run Program()` are presented, to demonstrate all of its various options and modes.

Introduction

The JMP Scripting Language includes a powerful feature that allows you, the script author, to launch other executable programs on the same machine that is running JMP. And not only can you launch these other programs; you can also send data to them and retrieve their output — all from your script. This feature is packaged as the built-in function `Run Program()`.

But great power often requires greater complexity. And `Run Program()` is indeed complex. For starters, it returns one of three different types of results, depending on how you call it. One of those result types is a JSL object to which you send additional messages, allowing you to control the executable program you launched. Another form lets you to embed JSL callback functions right in the call to `Run Program()` itself!

Syntax Overview

Let's start with the syntax of `Run Program()`.

```
Run Program(
     Executable( path to executable ),
     Options({ list of arguments }),
     Read Function( expression ),
     Write Function( expression ),
     Parameter( expression )
)
```

All the arguments to `Run Program()` are named, which is helpful because there are five of them. The first two are pretty straightforward. `Executable()` is used to provide the path to the executable you would like to launch. For JMP running on Macintosh[1], this must be the full path to the executable. The `Options()` argument lets you specify command line arguments for the executable.

[1] In this paper, I show all examples on the Macintosh. But `Run Program()` works equally well in JMP on Windows, although there are necessarily some differences. Consult the documentation for details.

For example, I might issue this ping command in Terminal.

```
machine:~ user$ ping -c 8 www.jmp.com
PING www.jmp.com (149.173.156.116): 56 data bytes
64 bytes from 149.173.156.116: icmp_seq=0 ttl=253 time=0.777 ms
64 bytes from 149.173.156.116: icmp_seq=1 ttl=253 time=0.657 ms
64 bytes from 149.173.156.116: icmp_seq=2 ttl=253 time=2.103 ms
64 bytes from 149.173.156.116: icmp_seq=3 ttl=253 time=1.831 ms
64 bytes from 149.173.156.116: icmp_seq=4 ttl=253 time=0.630 ms
64 bytes from 149.173.156.116: icmp_seq=5 ttl=253 time=0.625 ms
64 bytes from 149.173.156.116: icmp_seq=6 ttl=253 time=0.859 ms
64 bytes from 149.173.156.116: icmp_seq=7 ttl=253 time=0.902 ms

--- www.jmp.com ping statistics ---
8 packets transmitted, 8 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.625/1.048/2.103/0.543 ms
```

The first word of the command `ping` is the executable. The rest of the line are options. In Terminal, I can find the full path to the `ping` executable using the `whereis` command.

```
machine:~ user$ whereis ping
/sbin/ping
```

So my Run `Program()` script looks like this.

```
 Run Program(
    Executable( "/sbin/ping" ),
    Options({ "-c", "8", "www.jmp.com" }),
    Read Function( "text" )
 );
```

The tricky parts to remember are that `Executable()` must be a full path, and that `Options()` is a list of strings.

But the Read `Function("text")` part is a bit mysterious. This is actually how you specify what Run `Program()` returns. Putting Read `Function("text")` as we've done here causes Run `Program()` to return the output of the executable as a string[2]. When you run the JSL snippet above, JMP waits for `ping` to complete, then returns all of the output as a string that is displayed in the JMP Log.

You can also specify Read `Function("blob")`, which is very similar but returns a blob (binary large object) instead of a string. This is handy if your executable produces binary output, but it is beyond the scope of this paper.

If you don't specify Read `Function()` at all, or you specify it as a JSL function, then Run `Program()` returns a special *Run Program Object*. This is a special JSL object that lets you communicate with the executable. You can't really do much with a *Run Program Object* except

[2] In Unix parlance, both stdout and stderr are returned.

send it messages and assign it from one JSL variable to another. When you assign it to another variable, both variables just hold a reference to it. (This is the same way that references to a data table work in JSL.) But when the last reference to a *Run Program Object* goes away, JMP immediately terminates the executable.

When you specify `Read Function()` as "text" or "blob", JMP waits until the executable completes before returning from `Run Program()`. But when you specify `Read Function()` so that `Run Program()` returns a *Run Program Object*, JMP does not wait for the executable to finish but instead starts executing the next statement in your script immediately.

Using a Run Program Object

`Run Program()` with a *Run Program Object* is very powerful because your script can continue to run simultaneously with the `Run Program()` executable. Let's extend our `ping` example to use a *Run Program Object*. That way, we can print the output from `ping` while its running, rather than waiting until it completes.

```
host = "www.jmp.com";
count = 4;

// Define a function that will receive output from the ping
// executable
ping reader = Function( {ping},
   Write( "\!N" );
   Write( Trim( ping << Read ));
);

// Launch the ping executable, using ping reader as our
// Read Function()
ping = Run Program(
   Executable( "/sbin/ping" ),
   Options({ Eval Insert( "-c ^count^" ), host }),
   Read Function( ping reader )
);

// All done!
Write( "\!NDone!" );
```

When we run this JSL, we see the output from `ping` appear on the JMP Log as it is produced by the `ping` executable and received by JMP. Let's break this down.

- We parameterized the host and the ping count. We combined the `-c` and the count into a single option value (`Run Program()` doesn't seem to be too picky about this), and used the handy `Eval Insert()` function to build it.

- For our `Read Function()` we specified a function we created and named `ping reader`. JMP will automatically call this function when the `ping` executable has output to send us. And because we specified the `Read Function()` this way, we also caused `Run Program()` to return a *Run Program Object*.

- We assigned what `Run Program()` returned — the *Run Program Object* — to a variable, which we can think of as representing the executable. To help make that mental connection, I find it handy to name the variable after the executable, like we did here. If we hadn't assigned the *Run Program Object* to a variable, there would be no reference to it and JMP would terminate the `ping` executable immediately after creating it.

As we step through the execution of this JSL a bit more closely, we see that the call to `Run Program()` launches the `ping` executable and then immediately returns. Then the next line of our script is executed, and "Done!" is written to the JMP Log. At this point, our script has finished.

However, the *Run Program Object* lives on in our global JSL variable named `ping`. So the executable keeps running simultaneously with JMP. When it produces some output, JMP calls our `ping reader` function, passing it another reference to the same *Run Program Object* as a function argument, which we also named `ping` because we really like that name.

Since `ping reader` is only called when the executable has output ready for us to read, we can send the **<< Read** message to the *Run Program Object* to retrieve that output. In this example, we just write the output to the JMP Log. It looks like this:

JMP Log:

```
Done!
PING www.jmp.com (149.173.156.116): 56 data bytes
64 bytes from 149.173.156.116: icmp_seq=0 ttl=253 time=0.656 ms
64 bytes from 149.173.156.116: icmp_seq=1 ttl=253 time=1.072 ms
64 bytes from 149.173.156.116: icmp_seq=2 ttl=253 time=0.613 ms
64 bytes from 149.173.156.116: icmp_seq=3 ttl=253 time=0.680 ms

--- www.jmp.com ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.613/0.755/1.072/0.184 ms
```

Doing More With The Output

Let's continue to build on this example, and see what else we can do with the output from the `ping` executable.

```
host = "www.jmp.com";
count = 20;

// Create a data table to receive our data
dt = New Table( "ping",
   New Table Variable( "Host", host ),
   New Column( "icmp_seq" ),
   New Column( "ttl" ),
   New Column( "time", Set Property( "Units", "ms" ))
);

// Run ping and collect results
ping = Run Program(
   Executable( "/sbin/ping"),
   Options({ Eval Insert( "-c ^count^" ), host }),
   Parameter( dt ),
   Read Function(
      Function( {ping, dt},
         // Break read data into lines and process each
         l = Words( ping << Read, "\!n" );
         For( i = 1, i <= N Items( l ), i++,

            // Break line into words
            w = Words( l[i] );

            // Attempt to parse
            Try(
               v1 = Num( Words( w[5], "=" )[2] );
               v2 = Num( Words( w[6], "=" )[2] );
               v3 = Num( Words( w[7], "=" )[2] );

               // Add one row of data
               dt << Add Row({
                  :icmp_seq = v1;
                  :ttl = v2;
                  :time = v3; });
            ,
               // Log and skip lines that don't parse
               Write( Eval Insert( "\!N^l[ i ]^" ));
            );
         );
      )
   )
);

// Wait for ping to finish
While( !(ping << Is Read EOF), Wait( 0.1 ));

// Create a distribution of the ping times
dt << Distribution( Continuous Distribution( Column( :time ),
   Horizontal Layout( 1 ),
   Vertical( 0 )));
```

This script obviously does a lot more.

- For starters, it creates a new data table. We will collect the output of the `ping` executable into this data table.

- Our Run `Program()` JSL function is now using the `Parameter()` option; and our Read `Function()`, which we wrote in-line this time, has a second argument. These two changes are related. When you call Run `Program()` with the `Parameter()` option, you specify a single JSL variable that will later be passed to your Read `Function()`. In this case, we pass the reference to our data table.

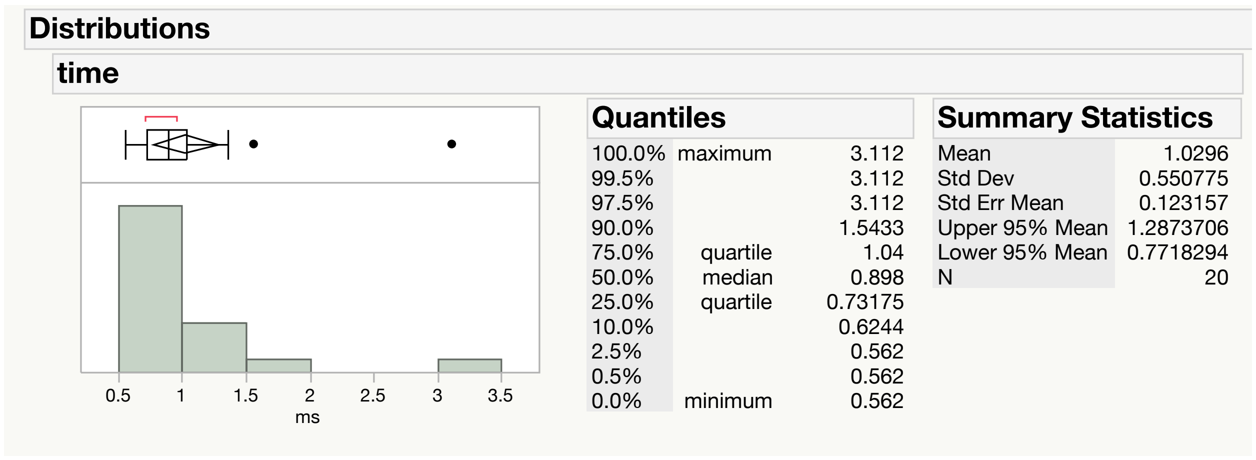
- In our Read `Function()`, we read output from the `ping` executable and then attempt to parse it. We are looking for the lines that have the icmp_seq, ttl (time to live) and time values. If we successfully parse these values from a line of output, we add them as a new row in our data table.

- For the rest of the output, we just echo it to the JMP Log.

- After we call `Run Program()` and launch the `ping` executable, we must wait for it to finish. We do this with a `While` loop that continuously sends our `ping` *Run Program Object* the **<< Is Read EOF** message. As long as this message returns false, we wait briefly and try again.

- Once the `ping` executable has finished, we perform a Distribution analysis on our collected times.

When I run this script on my machine with my network connection, I see results like this.



Distributions

time

Quantiles

100.0%	maximum	3.112
99.5%		3.112
97.5%		3.112
90.0%		1.5433
75.0%	quartile	1.04
50.0%	median	0.898
25.0%	quartile	0.73175
10.0%		0.6244
2.5%		0.562
0.5%		0.562
0.0%	minimum	0.562

Summary Statistics

Mean	1.0296
Std Dev	0.550775
Std Err Mean	0.123157
Upper 95% Mean	1.2873706
Lower 95% Mean	0.7718294
N	20

JMP Log:

```
PING www.jmp.com (149.173.156.116): 56 data bytes
--- www.jmp.com ping statistics ---
20 packets transmitted, 20 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.562/1.030/3.112/0.537 ms
Distribution[]
```

Sending Data To An Executable

`Run Program()` also lets us send data from JMP to our executable's standard input (stdin). But to demonstrate this, we need to use an executable other than `ping`, because `ping` doesn't read data from its stdin. Instead, we will use the Unix word counting program `wc`. The `wc` executable reads data from its stdin; then it writes the counts for the number of characters, words, and lines it received to its stdout.

In this example, we will first launch the `wc` executable. Then we will construct some text to send, line-by-line, through its *Run Program Object*. After waiting for `wc` to complete, we can read the final counts and display them.

```
      // Run the word count program, writing data to its stdin
      // and reading the results from its stdout
      wc = Run Program( Executable( "/usr/bin/wc" ));
```

```
// Get the script of Big Class as an array of lines
dt = Open( "$SAMPLE_DATA/Big Class.jmp", Invisible );
sb = Script Box( Char( dt << Get Script ) );
Close( dt );
sb << Reformat;
text = Words( sb << Get Text, "\!r" );

// Write the text to our program, then signal that we're done
For( i = 1, i <= N Items( text ), i++,
   wc << Write( Eval Insert( "^text[ i ]^\!n" ));
);
wc << Write EOF;

// Read the output from our program into a string
result str = "";
While( !(wc << Is Read EOF),
   If( wc << Can Read,
      result str ||= Trim( wc << Read )
   ,
      Wait( 0.1 )
   )
);

// Show the result
{lines, words, bytes} = Words( result str );
Show( lines, words, bytes );
```

Here's the breakdown.

- The very first thing we do is call `Run Program()` to create a *Run Program Object* for our `wc` executable. The `whereis` command tells us that the full path to `wc` is `/usr/bin/wc`. There are no options to specify, and we want the *Run Program Object* to be returned. So this is all we need.

- Now we need some text to send to `wc`. We will use the table script for the Big Class sample data table. This is a script JMP can generate that fully describes the data table. We use a `Script Box` to format it into multiple lines with nice indentation. Then we use the `Words()` function to break it up into an array of text, with each array element containing a single line.

- Next, we send our text to the `wc` executable using the **<< Write** message. That's easy enough, but we have to be sure to put a Mac newline at the end of each line of text; otherwise, `wc` won't recognize our line endings.

- When we've sent all the text, we tell `wc` we're done by sending the **<< Write EOF** message. This closes `wc`'s stdin and allows it to start tallying up its counts.

- We need to wait for `wc` to finish its work and write its answers to stdout. But instead of embedding this within a `Read Function()`, we do it in the main body of our script to better control the sequence of events. We use the same style of While loop as before. But inside, we first ask if `wc` has any data for us by sending the **<< Can Read** message. If it returns true, we **<< Read** what's available; otherwise, we wait a bit as before.

Calling **<< Can Read** isn't strictly necessary in this case, since **<< Read** will wait until data is available anyway. But it serves to demonstrate how this message can be used.

- Once we've gotten `wc`'s result, we parse it out and display it in the JMP Log.

JMP Log:

```
lines = "1188";
words = “1925";
bytes = “18537";
```

Conclusion

JMP's `Run Program()` feature extends your reach with JSL to any executable available on your system. By using a *Run Program Object*, you can send and receive data from executables you launch in a controlled and efficient manner, while the rest of your script continues to do useful work.

Now your scripting is not limited to only the capabilities of JSL. Anything your computer can do — image processing, audio conversion, process management, you name it — can be performed by JMP.

Contact information

Your comments and questions are valued and encouraged. Contact the author at:
Michael Hecht
michael.hecht@jmp.com